DB Server Project

Jim Kowalkowski, Anil Kumar, Marc Paterno, Steve White

1 Introduction

The purpose of this document is to define a set of the requirements for the dbservers and describe how the most important ones will be realized. The description includes high-level comments about the design goals and problem decomposition. This document should be considered a work in progress; reflecting the current view of the system as it is evolving. Each section describing a component will include the following information:

- *Design objectives*: an informal discussion of concepts and how the objects in the package will interact and the behavior they must exhibit.
- *Objects defined*: A list of all the objects that exist within the package with a brief explanation of what they do.
- *Configurable parameters*: All the configurable parameters for the package and a description of what they mean.

- *Call scenarios*: UML sequence diagrams (in time) illustrating the intended use patterns for the objects contained in this package. The diagrams will be included in the appendix.
- *Integration and testing plans*: Special requirements for testing or evaluating this package. If simulation is required to characterize an object's behavior under load, it is described here.

- *Events posted*: Each package is expected to produce event messages used for monitoring and performance analysis. This is a description of all the events posted by this package and the data they will contain.
- *Development effort*: A rough estimate of the amount of effort that will be required to complete the package.

In addition, each component will have a series of UML static class diagrams showing essential design elements such as relationships, classes, and methods. Many of the sections are expected to contain brief, unformatted contents, which will be modified as the design and implementation are made firm. (***Please note that the call scenarios are not complete yet***.)

Two new concepts are present in the above list: configurable parameters and events posted. Every configurable parameter will be changeable through an IDL interface dynamically (as the program is running). The system will provide a facility to manage and present the parameters to clients in a uniform way. This facility will also manage the posted events and is described in detail later in this document.

All work estimates are in man-days. Most of the design time (excluding the monitoring/control package) has already occurred. The numbers represent multiple persons working on the design.

2 Requirements

The requirements below are taken directly from Vicky's original document and labeled as (Orig-X), where X is the original numbering in that document. I've introduced a new numbering system in the form (R-X) where X is a number that indicates the requirement number and the relative priority or the requirement. I broke several of the requirements into more than one and just added a letter to indicate the requirement within the group. Some of the requirements are missing actual timing or size values. Those values will need to be filled in before we can know if the requirement can be accomplished or not.

R1a	The server shall allow for concurrent transactions. This will increase server throughput and lower the waiting time for smaller transactions (decrease queue time).
R1b	The total number of active database connections in a single number instance shall be restricted to a number configured by the user. These connections will be disconnected after a configurable duration of time to minimize the database resources consumed by the server.
R2	The output of the code generators shall only be files unique to the application being built. The files common to all DB servers shall exist in one central location and referenced by the generated files. The application shall be rebuilt when dependent have been updated.
R3	The DB server shall keep operating statistics that can be accessed from clients.
R4	The DB server must be able to identify the following for each request: location (client machine), who (the Unix user name or equivalent), and the process name/ID. The DB server shall use this information when logging problems.
R5a	The DB server shall deliver constant sets of small size to the client in a reasonable time (average over accesses of the same set). The DB server shall deliver constant sets of large size to the client in a reasonable time (average over accesses of the same set). The first access of a new set (not seen before by the server) can take longer than subsequent accesses. This requirement is considered complete. Note that small, large, and reasonable are concrete enough to really complete this requirement.
R5b	Application other than calibration d0om servers should be investigated to see if R5a can be applied to them also.
R6	The DB servers shall restart if they fail, shall reconnect to the database if the database fails. A failure shall be accompanied by information that allows the responsible party to locate the source of the failure back to the originator of the request (if this is the source of the failure).
R7a	DB servers shall handle mixed case text the same way
R7b	DB servers shall have a common security service available for use.
R7c	DB servers shall provide a mechanism for delivering common exception to the clients
R8	The ability to run a local naming service shall be provided. This naming service shall keep forward requests to a parent name server (global

	server).
R9a	The DB server shall make a subset of read-only data available to clients even if the remote database goes down.
R9b	The DB server shall guarantee execution of requests (up to a user configuration time limit), regardless if the database is up or down (desirable feature).
R10	The DB server shall transparently handle reconnecting to a "hot replica" of the current database in the event of a main database failure without affecting the connected clients.
R11	The server shall have a proxy mode, where is connected to another server that actually executes the transactions on its behalf.
R11	Clients shall only make retain a user configurable number of calibration "tree" to minimize the accesses to the server and reduce transfer times.
R12	Nothing to restate. This cannot be required.

Draft

3 Original Requirements Translated

(Orig-1) Client code needs to obtain services from a database server without waiting in line behind other clients who may have requested a service that initiates a lengthy database query or transaction. However, the total number of database connections within one server needs to be restricted.

(R-1a): The server shall allow for concurrent transactions. This will increase server throughput and lower the waiting time for smaller transactions (decrease queue time).

(R-1b): The total number of active database connections in a single number instance shall be restricted to a number configured by the user. These connections will be disconnected after a configurable duration of time to minimize the database resources consumed by the server.

Current System Support: No.

Note: Use a multithreaded ORB (OmniORB). Make the application multithreaded. The increased server throughput can be viewed as partially only a perception in Python, since multiple threads do not span processed.

Note: Create a pool of database connections that are acquired and release, and maintained in last-time-used order. Scan the connection list every so often to remove the idle connections.

Note: A thorough understanding of the current design is necessary to evaluate whether or not these things can be retrofitted.

(Orig-2) Some features that are not provided in the generated core db server code have been identified and implemented in ad-hoc ways in different applications. They need to be moved into the core generated code layer of the db server. Examples of these are

- Handling mixed case text

- Handling restricted access to certain services, based on users and roles (security)
- Handling exceptions in a way that is easy for the client to deal with

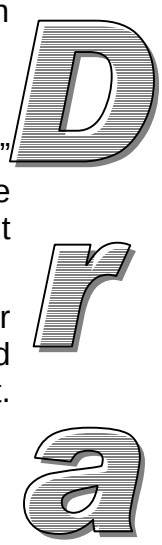
(R-7a): DB servers shall handle mixed case text the same way

(R-7b): DB servers shall have a common security service available for use.

(R-7c): DB servers shall provide a mechanism for delivering common exception to the clients

Current System Support: Yes for a and c, no for b.

Note: Exception processing needs investigation to determine what "easy" means and what is acceptable for everyone. Mixed case text is available in the server code, but not used at D0. Security requirement were discuss at a recent meeting; discussed using Unix login information to determine roles.



(Orig-3) Some clients need to be able to communicate with a database server that does not itself have an open connection to a database, but instead communicates with and passes on its work to another db server that services it. This may be highly desirable for two reasons

- Ease of deployment – not requiring oracle_client code to be installed.
- Management of database connection resource – must limit total number

(R-11): The server shall have a proxy mode, where is connected to another server that actually executes the transactions on its behalf.

Current System Support: No.

(Orig-4) Some clients run on nodes that do not have network access to the outside world, but only to their intranet, one or more nodes of which is able to access the wider internet. Such clients are obvious candidates for the services in item 3). Along with this comes the need to use a locally accessible naming service for looking up servers. Although this problem is not unique to database servers it must be solved for them. Either a distributed naming service or a similar approach to 3) needs to be used to forward naming service requests from a local naming service to the central one.

(R-8): The ability to run a local naming service shall be provided. This naming service shall keep forward requests to a parent name server (global server).

Current System Support: No.

Note: The ability can likely be isolated into a separate project.

-

(Orig-5) Clients may need to obtain partial services from a database server even when connection to a central database is not available. Different applications need to study and understand where they could make use of partial services. SAM is a prime example where partial services could be used to good advantage to improve robustness and reliability of the system. Examples of desirable partial services are

- Providing server-side cached results for read-only data when database access is unavailable
- Queuing and retrying a database write request when database access is down – with guarantee of eventual execution of request.

(R-9a): The DB server shall make a subset of read-only data available to clients even if the remote database goes down.

(R-9b): The DB server shall guarantee execution of requests (up to a user configuration time limit), regardless if the database is up or down (desirable feature).

Current System Support: Partial support for 9a – objects are cached in memory. Not supported for 9b.

Note: It may be difficult to queue many requests (or even one). Communicating results (success/failure) is difficult when the application may have gone away. Both of these can probably be an addition to the system.

(Orig-6) For some applications it may be necessary to implement a 'hot replica' of the database in order to fail-over to this database. Database servers need to be able to receive instructions to re-initiate their connections to a designated database.

(R-10): The DB server shall transparently handle reconnecting to a "hot replica" of the current database in the event of a main database failure without affecting the connected clients.

Current System Support: No

Note: Are "hot replica" locations predefined at program startup time? This can be viewed as an addition to the system.

(Orig-7) Some applications, such as those requiring hundreds of thousands of calibration constants to be fetched, need the load and fetch of those constants to be done as rapidly as possible, and certainly more rapidly than the current infrastructure provides for. Since these sets of constants consist of read-only constants, server side caching strategies may be considered without fear that cached data might become inconsistent with the primary copy of the data in the database. Work must be done to improve performance of both of the following for calibration servers

- Loading of constants into db server memory
- Serving up of constants to the application

(R-5a): The DB server shall deliver constant sets of small size to the client in a reasonable time (average over accesses of the same set). The DB server shall deliver constant sets of large size to the client in a reasonable time (average over accesses of the same set). The first access of a new set (not seen before by the server) can take longer than subsequent accesses. This requirement is considered complete. Note that small, large, and reasonable are concrete enough to really complete this requirement.

Current System Support: Consider met.

Note: Good database table design and caching strategies are needed to achieve this. A good use of CORBA is probably going to be needed (efficient delivery of all required data).

(Orig-8) In memory caching of results is only currently implemented for d0om objects. Other applications may require and benefit from both server in-memory caching and server-side caching and this should be an optional feature available in the core services of a db server.

(R-5b): Application other than calibration d0om servers should be investigated to see if (Orig-7) can be applied to them also.

Current System Support: Current system provides in memory caching of d0om objects.

Note: See (Orig-7). The addition thing implied here is that a local file system cache may be necessary to achieve the required timing numbers.

(Orig-9) Monitoring and debugging of the system is hampered by being unable to identify from where a server request has come. All IDL for database servers, including access to the core services in all db servers, needs to provide for a user context object that identifies the caller and caller's location and potentially other information about the callers environment (version, etc.). This needs to show up in log files.

(R-4): The DB server must be able to identify the following for each request: location (client machine), who (the Unix user name or equivalent), and the process name/ID. The DB server shall use this information when logging problems.

Current System Support: No.

Note: Means changes to the IDL and client code.

(Orig-10) Access to timing and performance information tracked by a server needs to be made a standard service of a db server

(R-3): The DB server shall keep operating statistics that can be accessed from clients.

Current System Support: Processing of log times only, external to the server process.

Note: none.

(Orig-11) Application developers need to have improved procedures for building their application and db server. This must allow for keeping each application up to date with evolving db server infrastructure changes, without sending mail asking each developer to perform a sequence of manual copies or other operations. Somehow the dependency on core infrastructure modules must be built in formally – so that a build procedure picks up any improved infrastructure

modifications (apart from via the obvious method of getting changes propagated via db_server_gen).

(R-2): The output of the code generators shall only be files unique to the application being built. The files common to all DB servers shall exist in one central location and referenced by the generated files. The application shall be rebuilt when dependent have been updated.

Current System Support: Partial. Redundant files currently exist.

Note: This may require the addition of inheritance layers or new abstractions.

(Orig-12) d0om-corba and calibration manager code used in client applications do not currently provide for client-side caching of multiple calibration 'trees' or of subparts of those 'trees'. It needs to be understood, by study of real use-cases, whether much would be gained by such client-side caching strategies.

(R-11): Clients shall only make retain a user configurable number of calibration "tree" to minimize the accesses to the server and reduce transfer times.

Current System Support: None.

Note: This is difficult to address if it involved D0OM or requires changes to existing calibration code. Herb and others will need to be consulted here to find out if this is possible and where it can be added. Is there any benefit of caching the information as a tree in the client? Can the server object be located upon first access to the data and the entire tree loaded at once? Is there a case where only part of the tree is accessed?

(Orig-13) The current db server core infrastructure needs to be gone over with an eye to robustness – ensuring that restarts work every time, that reconnection to the database is initiated after a failure, that core dumps of a server can be diagnosed and the root cause problem found and fixed. This is an application requirement – that every database server is up all of the time and recovers from all types of outages.

(R-6): The DB servers shall restart if they fail, shall reconnect to the database if the database fails. A failure shall be accompanied by information that allows the responsible party to locate the source of the failure back to the originator of the request (if this is the source of the failure).

Current System Support: Some. The DB servers restart.

Note: none.

(Orig-14) Web based (and other) applications would benefit from having db servers that support an alternate protocol to CORBA IDL/IIOP. An XML data format using http (for example), or other similar standards-based protocol would be desirable. This would involve extensions in code generation as well as an IDL to XML (or ?) translator. This may not be essential for D0, but might prove essential for future Grid work, or for other experiments that might benefit from this database infrastructure.

(R-12): Nothing to restate. This cannot be required.

Current System Support: No.

Note: This is interesting to think about.

4 Architecture Overview

R1 has the largest impact because it changes the design and therefore existing infrastructure. The initial set of tasks will avoid changes in the client and business logic and only address lower level code in the server. Concurrency (through threading) cannot be achieved without refactoring many of the large classes and without making use of encapsulation. The server system will not be easily maintained or expandable without refactoring and testing. In order to accomplish R1 and R3, we will break them down into the steps outlined in the following sections. The initial plan it to attack the problem from the bottom up. Each section will follow a development cycle that includes the following items in the order that appear here.

- *Design*. Including UML static class diagrams and sequence diagrams that capture the essential call patterns. We only want to capture essential elements of the design, not every detail.
- *Test Programs*. Component test program will be written and integrated into the automated test suite that validates components. The test programs will test for survival in a multi-threaded environment – preferably in a deterministic way (as opposed to random chance).
- *Implementation*. The components (classes) will be written and not complete until the test programs run.
- *Integration Testing*. These programs will verify that the server is function as a whole, as promised. This will likely require load testing and may require some kind of simulation.

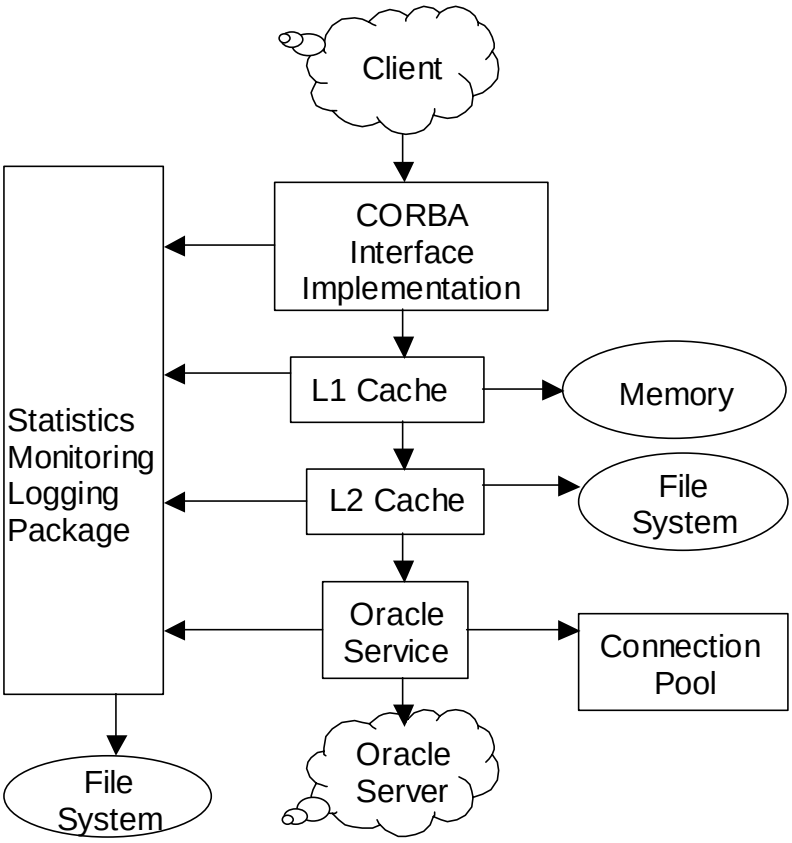


Figure 1: Simplified architecture view

5 Addressing R1 – Threading

Here we will concentrate on the lower level tools that the system provides. Design changes will include breaking up the classes so each new class performs one task and has an associated component test. Each component test will insure that the object will survive in a multithreaded application. It is important to note that completion of R1 will allow the proxy server feature to be easily added. The changes will be designed to accommodate this functionality.

5.1 Database Access

5.1.1 Database connection management

Concurrency will be managed at the database connection level. Requests from threads for a connection will be handled in the order in which they are received from a connection manager service. The actual DB connection will be wrapped in a class that monitors the request time, the acquire time, and the release time. A server will be started with a limited number of concurrent connections allowed to the database. The timing values will be sent to the monitoring/control package for proper handling during a connection release. Each connection will be used exclusively by the thread that acquired it. The connection management service will periodically scan all the connections to close down idle connections and send

alarms for connections that are held too long by client threads. The connections will eventually be capable of reconnecting to a database if the database disconnects. A policy for handling transactions currently in progress will not be addressed in this phase of development.

5.1.2 Database transaction control and execution

A connection can only be held for the duration of a method call. At the beginning of the call, a connection is acquired and at the end of a call the connection is released. An SQL statement execution object will be constructed using a connection. The lifetime of these objects is expected to be only as long as the connection is held. Many SQL statements can be executed during the lifetime of an SQL execution object. Each method of the SQL executor will send timing information to the monitoring/control package for processing.

5.1.3 Objects and Interfaces

(*to be completed – include exceptions and component test plan for each object may be able to refer directly to the documentation in the Python files if that is the way that it will be maintained, a high level description could be maintained here also*)

- *ConnMgr*: The thread safe pool of connections. Users will acquire and release connections using this class.
- *ConnScanner*: A separate thread that monitors the connections. It watches for connections exceeding configured idle time and held time.
- *Connection*: A connection to the database. May be in a disconnected state.
- *SQL*: An executor of transactions (queries) on a connection.

5.1.4 Configuration Parameters

(*to be completed – include full description*)

- Number of connections in the pool
- Idle connection shutdown length
- Scan rate of idle connections
- Database to connect to (user/password/node)
- Database availability (up/down)

5.1.5 Integration Test Plan

(*to be completed – include load testing and simulation if applicable*)

It may be worthwhile to produce a test program that will measure connection queue effects. This load-testing program should take three parameters: number of threads, number of available connections, and number of rows to return. The test should run through many combinations of these three parameters and measure the time per transaction for each case. The number of rows should be controlled by a simple where clause. The table accessed should contain dummy data and select the group of rows to access randomly.

5.1.6 Call Scenarios

(*to be completed – simple UML sequence diagram showing basic call patterns*)

5.1.7 Events Posted

(*to be completed*)

- ReleasedConnection: time_requested, time_acquired, time_released, pending_queue_length, idle_connection_count, number_of_threads
- ConnectionsState: pending_queue_length, idle_connection_count, number_of_threads
- TransactionEnd: total_time (tagged with description of the transaction type)

5.1.8 Development Effort

(*to be completed*)

design=8, implement=7, test=2, integration=3, verification=5

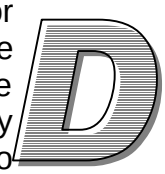
Design is 100% complete, implementation is 90% complete.

5.2 Caching Services

We will pursue a hierarchical caching strategy. The user code will always interact with a single object to get data using a key so that coordination of database activity and object management is clearly encapsulated. The request will work its way through layers to find the desired object: memory, file system, and finally the relational database. The current server does not operate in this manner; the cache includes many CORBA objects and activities involving the cache and database access are not well encapsulated. The cache will be another point where concurrent access is necessary and will be managed carefully. The cache will utilize a shared lock mechanism in order to maximize concurrent access to the objects it holds. Locking is necessary in a multithreaded environment to ensure that the data structures are consistent. Multiple cache readers are allowed at one time. A cache update (addition of an object or removal of objects) required exclusive access. Each cache access will send details to the monitoring package for proper processing. The cache will manage some collection of objects up to a maximum amount maintained in least recently used order (fixed insertion policy). When the maximum is reached, a used defined policy object will determine what to clean out of the cache. The default policy will remove a configured percentage of the oldest objects. The cache is designed to have multiple keys reference the same object. A decision was made that the cache size will be expressed in number of objects, as opposed to total memory used. The cost of determining the memory used is likely to be too great to be worthwhile. This decision requires an intelligent choice for the total objects cached configuration parameter. Another important aspect of the cache is that is will generate only one database transaction if many requests simultaneously demand the same object. The first to request the object will start the transaction, all others will be put into a wait state until the transaction completes, at which time they will be awakened.

The requirements differ for caching amongst the various applications. Applications with clients that access similar read-only data can make good use of caching services. This list obviously includes calibration servers and RCP servers and perhaps run summary servers. In order to make the cache useful among many of the applications, we may need to generalize it.

At the lowest level, it is likely that the application will need to distinguish between the key types or attributes that will be passed down. A simple factory or dispatching system will need to be in place to make this possible. The OracleSource will require the key to contain the table name and the where clause. The table name will determine what object needs to execute the query and format the results. The table name will be the key into the dispatch table to quickly locate the correct handling code.



5.2.1 Objects Defined

(*to be completed – include exceptions and component test plan for each object, could be link to python help output and maintained in the python class*)

- LockSet – Shared access for read, exclusive for write. Testing will be difficult and timing diagrams will be necessary along with a multithreaded test. We will need to decide if the lock needed to be deterministic in how it dispatches waiting clients. The algorithm needs to be verified with other sources (transaction processing text books).

- Policy – An object that gets invoked when the cache is filled and more objects need to be added. Its job is to clean out objects from the cache. The policy will be invoked with the list of cached objects. It will remove a number of objects and return the list of deleted objects to the caller.

- DataSource – An abstract base class representing an object that retrieves information given a key.
- MemorySource – A concrete DataSource that caches objects in memory.
- FileSource – A concrete DataSource that caches objects in disk files.
- OracleSource – A concrete DataSource that retrieves objects from Oracle. This object uses the existing code generated DB layer to execute queries (one object per table). This layer will use the Factory for dispatching the request to the correct DB layer object using the key's table name attribute.

5.2.2 Configuration Parameters

(*to be completed – include full description*)

- Maximum number of objects to cache
- Policy object for cleaning objects from the cache (default is percentage to remove)
- Next lower cache tier or database access object

5.2.3 Integration Test Plan

(*to be completed – include load testing and simulation if applicable*)

A cache access time study should be completed. A program that simulates many accesses to a populated cache should be written to evaluate the design choices (dictionary with string key for example). The simulation should build and measure access times given many combinations of the following parameters: number of objects, object size, number of threads, and keys for the objects. Too much uniformity in object size and key size may skew the test, so sizes may need to be generated from a normal distribution. Additional parameters of mean/sigma may also be needed for generating the random sample of keys and objects. The outcome should be a plot showing the expected time for various parameter values.

Draft

5.2.4 Call Scenarios

(*to be completed – simple UML sequence diagram showing basic call patterns*)

5.2.5 Events Posted

(*to be completed*)

- CacheRetrieval: time_started, time_lower_tier, time_ended, hit_miss_flag, busy_avail_flag, total_cached_objects, total_threads
- CacheFull: time_cleanup_start, time_cleanup_end

5.2.6 Development Effort

(*to be completed*)

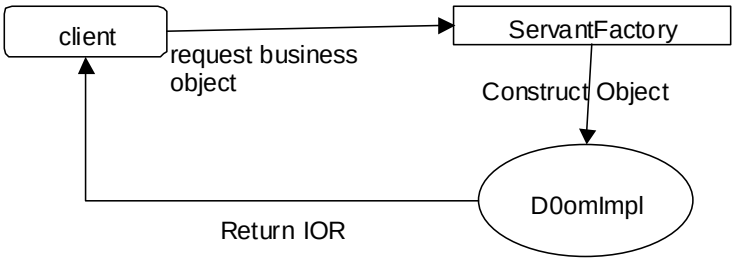
design=6+6, implement=14, test=3, verification=5

Design is about 70% complete, implementation about 10% complete.

5.3 Server Application Code – D0OM Interactions

This section will detail, by way of example, the proposed changes for server and d0omCORBA client communication. The goals are to stop server objects from being created before they are required, give the server total control of the cache, provide a way of identifying who is calling the server, allow for a “tree” of data to be written and read from a file, and interface properly with the requirements of multi threading.

5.3.1 Establishing Communications



The client must first establish communication with the server. As before the client will call createD0omServant, however at this point things change. This method will require identification data to passed in. The data will is expected to contain:

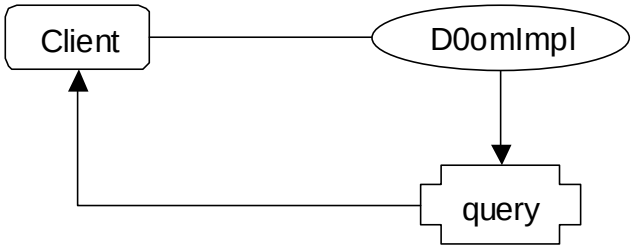
- Name of the node client is running on.
- The client's PID.
- Name of the user.
- Name of the application
- Arguments the user application was start with (argc/argv).

This information will be used by the statistics package currently under design and by DbLog.py

When a request is received by the ServantFactory it will create an instance of D0omImpl specific to that request. Unlike the current server, no two requests will ever share the same object. The instance of D0omImpl will be stamped with the identification data and its IOR returned to the client. The ServantFactory will not maintain any further knowledge of that instance. If the client inadvertently dies we expect the ORB to notice this and to delete the object.

5.3.2 Requesting Data from the Server

All communications between the client and the server will be through its specific instance of D0omImpl. There will be no other CORBA objects for the client to contact. The cache will no longer store CORBA objects as it is not possible to serialize them. It will store simple data objects which can be then be returned to the client (CORBA structs).



As show above, requesting data from the server requires the client to call its D0omImpl's query method. The query method will ask the memory cache for the data. If found, it will be returned to the client, otherwise it will go to a file cache (when implemented) and finally to the database itself. Note that one call immediately returns the data to the client. The getRef call will not be implemented on the server. This gives the server total control of its cache and if the server dies clients will not die when contacting the new server for data.

As in the current server, the query method requires the following attributes:

- Class name - cpp name of the table being queried.

- Condition – the where clause defined according to d0omCORBA standards.
- Tables – any additional tables added into the where clause
- Fields – any additional fields included into the where clause

The query method will return an object to the client which is similar but not exactly like the currently returned object. The difference is, pointers to CORBA objects will not be returned. Replacing that data will be a structure containing the name of the relationship, the object's name (cppObject), and the "where clause" which must be executed to produce data for that relationship.

When the client needs to move to the next level of the tree, it again calls the query method. It will pass in the object's name for the associated relationship, and the "where clause" it was given. This data will be used to get a cache hit and must not be altered by the client.

5.3.3 Terminating the Client

We expect that when a client terminates its run it will inform the server release its instance of D0omImpl. A simple method will be provided with which this can be done. For those clients who fail to obey the rules, we expect the ORB to catch the drop of the connection and to delete the object.

5.3.4 Interfaces Objects and Definitions

(*to be completed – include exceptions and component test plan for each object*)

5.3.5 Configuration Parameters

(*to be completed – include full description*)

5.3.6 Integration Test Plan

(*to be completed – include load testing and simulation if applicable*)

5.3.7 Call Scenarios

(*to be completed – simple UML sequence diagram showing basic call patterns*)

5.3.8 Events Posted

(*to be completed*)

- MethodCalled: time_start, time_end (tagged with method name)

5.3.9 Development Effort

(to be completed)

design=4+5, implement=5, test=3, integration=10, verification=15

Design is about 40% complete.

6 Addressing R3 - Monitoring and Control

The design of this package is still a work in progress and there are several design choices that still need to be made. It covers several concepts for which the definition of each is given below.

- *Monitoring*: Watching the current resource usage levels through polling or receiving event notifications asynchronously (registering for interested events).
- *Logging*: Verbose recording of interesting events (errors, informational or debugging).
- *Alarms*: Asking for asynchronous notification of an event only if the value associated with that event exceeds a given threshold. Event examples are resource levels or logging record for transaction timing. This feature will be very limited in the first version.
- *Statistics*: The fixed recording of important or interesting events for analysis purposes (determining proper resource allocations and load characteristics).
- *Control*: Changing resource allocations or parameters dynamically.

This package does not distinguish at the object level between the various types of events that can be generated; all look the same. The types of events are log, state, and statistical. All three of these types of events are managed differently and this package will have specific interfaces for handling them. Log message events are meant to capture out of the ordinary occurrences and can be filtered (configurable threshold). Statistical events are not filtered and are generated at higher volumes for each interesting thing occurring within a module or library. These events are meant for statistical analysis purposes. The state record events can to be generated at any time (on demand from other modules). They serve as snapshots of the current state.

The dbserver manages and makes use of many resources. In order to determine the correct resource allocations and understand the load characters of the dbserver, we need to collect statistics. The current proposal is for this package to contain input interfaces for other packages within the dbserver to add events and output interfaces for delivering the events to subscribers and to log files. We want to capture important server state information for each remote method invocation. Examples of some of the important attributes that we want for each remote method invocation include the following:

- WCT: wall clock time
- SEQ: sequence number representing the requesting client (who)
- HM: cache hit/miss flag
- BA: cache busy/available flag
- TT: total active threads in server
- COT: total cached object count
- QD: connection queue depth (pending connection requests)
- IC: inactive database connection count
- DBT: time that DB connection was held

- QT: time that thread was queued for a connection
- PT: remote method processing time.

Each package or module in the system interested in generating events or resource level events will register as a producer. Part of the registration will be identification of the event and a description of the variables in the record it will produce. Applications external to this process will have the ability to register (subscribe) with this dbserver as event recorders using a special IDL interface. This package will determine, based on configuration parameters and subscription information, where to route each event.

6.1 The Event

6.1.1 Identification (Header)

This module has to be efficient in it's handling of events and still have flexibility for selection and labeling. Flexibility means multiple string tags or names identifying a single event. Efficient handling of event in terms of processing and storage means that a single integer value should identify a type of event. In order to achieve this, we will produce an interface to register event types using complex strings that will be assigned a unique integer. This integer will be used as the identifier for each generated event. The verbose name of the event is still being determined. Two choices are: a name that consists of a list of strings, or a name that has a fixed number of strings that have specific meaning, for example:

- MOD: module that this event came from
- TYPE: type of event, similar to class (select/insert/update/array insert/etc.)
- INST: name of the event, similar to instance (table name for example).

Each variable within an event can be uniquely identified by an instance of class VariableID. A VariableID object is an event-id and a column-index within that event.

6.1.2 Body

The event body will be produced as a tuple or array of floats.

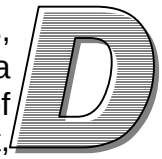
6.1.3 User Identification

Each remote user is assigned a unique object instance that they interact with. The servant factory (the single object that gives out the instances for the remote client) will be required to register a new user with this monitoring/control facility. The registration will return back a unique sequence number for this client. Each event generated must be tied to this sequence number to identify the underlying cause of the event. This event facility must be notified when clients disconnect so that the data associated with a sequence number can be cleared. One implication of this requirement is that objects that give out resources must be handed sequence numbers that represent the customer that is using the resource and the method they invoked.

6.2 Producers

Modules will describe the events that they will generate to the monitoring/control package. They must give the verbose name of the event and a description of the columns of the event. Events will contain a collection of floating point values. The registration interface will return the unique integer identifier for the event type.

6.2.1 Log Event Generation



Subsystems are expected to generate log events representing errors, warnings, and informational data. Each generated message should be associated with a particular statistical variable and a form string for producing a readable version of the event. Some examples of these variables are the time to execute a select, the length of the connection queue, or the number of objects in the cache. The first version of this package will implement a simple filtering mechanism for limiting the number of log events delivered to subscribers. The gathered statistical events (history) will be used to generate a series of probability distributions for each of the interesting variables. Log messages will be delivered to subscribers if the current measurement is further than n times away from the standard deviation for the current time (where n is a configurable parameter). Because all the history is used to determine the distributions, we will not notice slow changes or trends. This is not the purpose of this portion of the library. The purpose is to catch uncommon or strange interactions. It must be possible to add more filtering objects as they are needed. Therefore the system has to have a filter abstraction.

6.2.2 Statistical Event Generation

The modules are expected to post this type of event when the activity they represent has completed. An example would be the release of a connection. It should contain timing information about the particular action taken.

6.2.3 State Event Generation

For each pollable event type, a generator function will need to be registered. When invoked, this function is expected to produce a particular state event with the fields filled in. The monitoring module will demand the state events at any time. They should therefore not include any transitory measurements.

6.3 Remote Monitoring Applications (Consumers)

The remote monitor will subscribe to a server instance to receive event notifications. Application must be able to register for sets of events in addition to specific events. Registration will therefore be by the verbose string naming. If the array of names design is implemented, then any event that has any name that matches the subscription name will be posted (one to many mapping in this case). If the fixed names design is chosen, then if the subscriber asked for events from a particular module, it will receive all events generated by that module. The ability of the application to receive asynchronous notification of events means that the remote monitor application must be an ORB. Monitoring applications can attach to many servers and report resource usage and

problems. They can also be used to do global analysis for the purpose of making changes to resource levels in the running servers. These monitoring applications will likely need to be authenticated to make resource allocation changes and probably to receive events. A web based status display can be built using the facilities of this package. It is important to note that the monitoring interface allows for polling for the current state records and getting periodic delivery of these records.

6.4 Managing Events

There are several aspects to event handling besides the immediate delivery to remote monitoring applications. The events will be stored quickly into files. The files must contain a block of metadata that describes the event(s) held within. This format will not be compact. At a configured rate (daily), a job will be run that will transform the verbose files into a compressed, compact binary representation. This new format will be something that can be read by ROOT or another statistical analysis tool. The event writer will need to change files when they approach a configured maximum size.

In addition to the compacting job, we will supply a simple analysis job that produces the distributions necessary to perform the log event filtering. The assumption is that for each variable in a statistical event, there exists a distribution. Not only that, but the distribution is specific for a time interval, for example Mondays between 10am and 11am. The distribution will include all the history for that variable. A job will need to run periodically (once a week) to recalculate the distributions. It is essential that this function is quick to evaluate.

6.5 Configurable Parameters

(*this section will contain the plan for managing the configuration parameters advertised by each of the packages*)

6.6 Objects and Interfaces Defined

(*to be completed – include exceptions and component test plan for each object*)

6.7 Events Posted

(*to be completed*)

This package will produce a state event that contains rate information for each of the events that are posted by other packages. The rates will be in posts per hour. Each column will be a single rate measurement. The event descriptor will contain the event Ids associated with each measurement.

6.8 Configuration Parameters

(*to be completed – include full description*)

- Logging threshold value
- Maximum periodic event delivery rate
- Event file rollover

- Log file rollover
- Event analyzer rate
- Event analyzer process

6.9 Call Scenarios

(*to be completed – simple UML sequence diagram showing basic call patterns*)

6.10 Integration Test Plan

(*to be completed – include load testing and simulation if applicable*)

6.11 Development Effort

(*to be completed*)

design=5+7, implement=10, test=5, integration=7, verification=5

The design is about 30% complete.

7 Addressing R2 – Libraries and Code Generation

Requirement R2 will not be considered complete until the code generator only produces code that is unique to the package being generated. All other code must be available in a library.

8 System Verification

The system must be tested and performance measured as a whole. A reasonable way to accomplish this is through simple simulation. We are interested in measuring access times for information in the various server types. In order to do this, we will need a database loaded this data in a realistic fashion. The data does not need to be real information from the detector, random numbers will do just fine. The database can be populated in two ways: through the server or through SQL directly. Either way it requires understanding the rates and sizes for each of the tables that will be accessed during the simulation. The size means number of rows in this context. A large enough sample must be present in the database for the simulation to be meaningful (six months to a one years worth of data for example). The read speed measurement must be taken under full load. This means starting many jobs that mimic the behavior of the various use cases we are familiar with: local and remote analysis jobs (handful of events per runs), remote and local farm processing (production). The simulation must not require real detector data and will not require running a reconstruction program.

9 Notes

9.1 Current System

A set of UML static class diagrams for the current dbserver may be necessary to understand what the current server does and all its interactions.

The existing code existing code needs to be cleaned up. The class level attributes need to be removed. Method calls should never create new attributes in the class.

9.2 Areas not Addressed

The calibration writing code will be left intact. This is considered to be a specialized task and the function may be separated out into its own server.

9.3 Areas to be left as is for now

The current exception processing strategy will be left as is. The new components may generate new exceptions.

The current logging facility will be left intact along with the messages it can currently contain.

9.4 Backward Compatibility

The DbCore class interface will remain intact. The underlying code will now be implemented using the new SQL and connection classes. The connection held by this object will remain for the lifetime of the DbCore object.

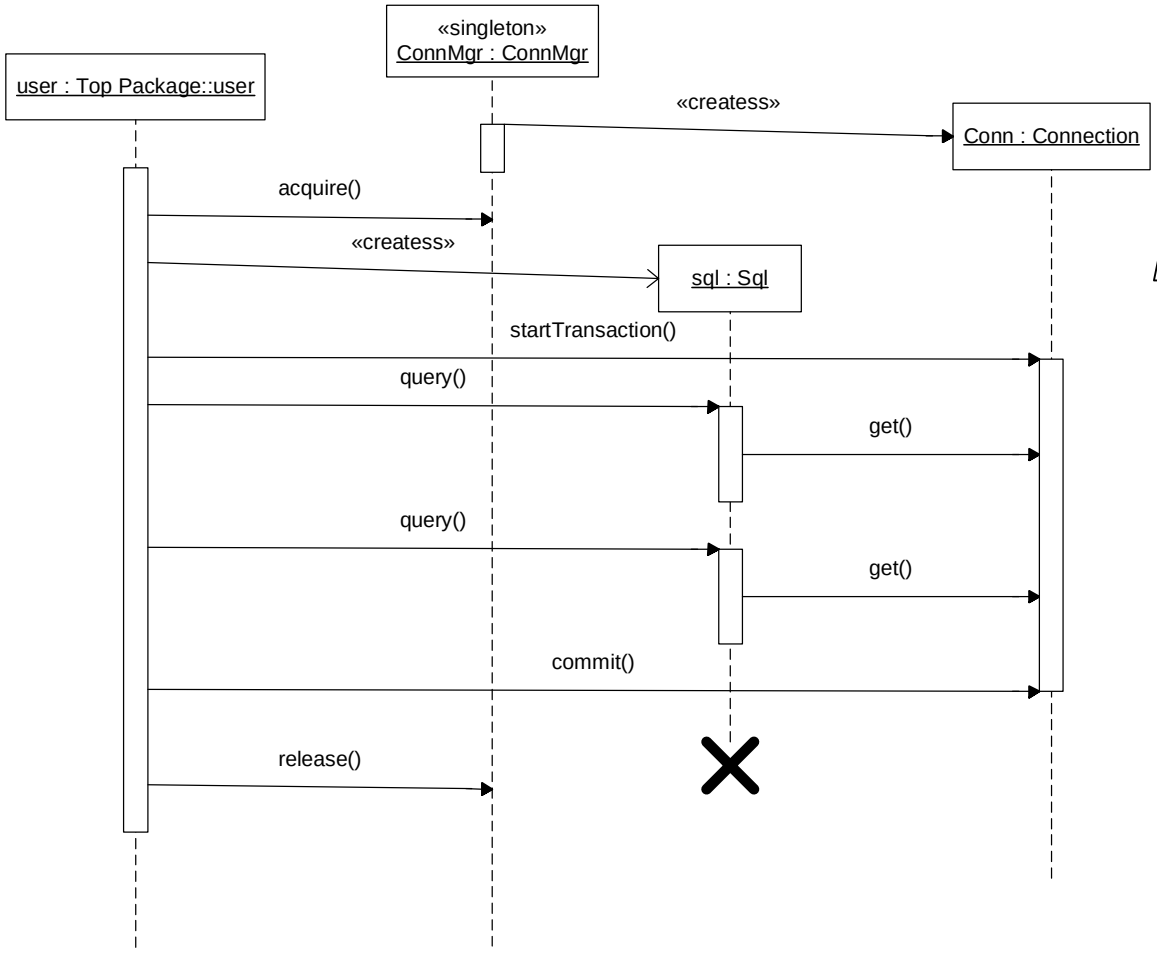
9.5 Proxy Server

This document uses the term proxy server. The functioning of the dbserver outlined here is not a true proxy. A true proxy would attach itself to another dbserver downstream and forward messages to it. The initial implementation described here can be thought of as a dbserver with a persistent cache. This dbserver will attach, at the lower level, directly to the remote Oracle database, as opposed to another remote dbserver. We do not address the development effort required to forward messages to a remote dbserver. The caching system developed here will allow a concrete DataSource to be developed that can attach to another server to retrieve information to achieve a true proxy service.

9.6 To be Determined

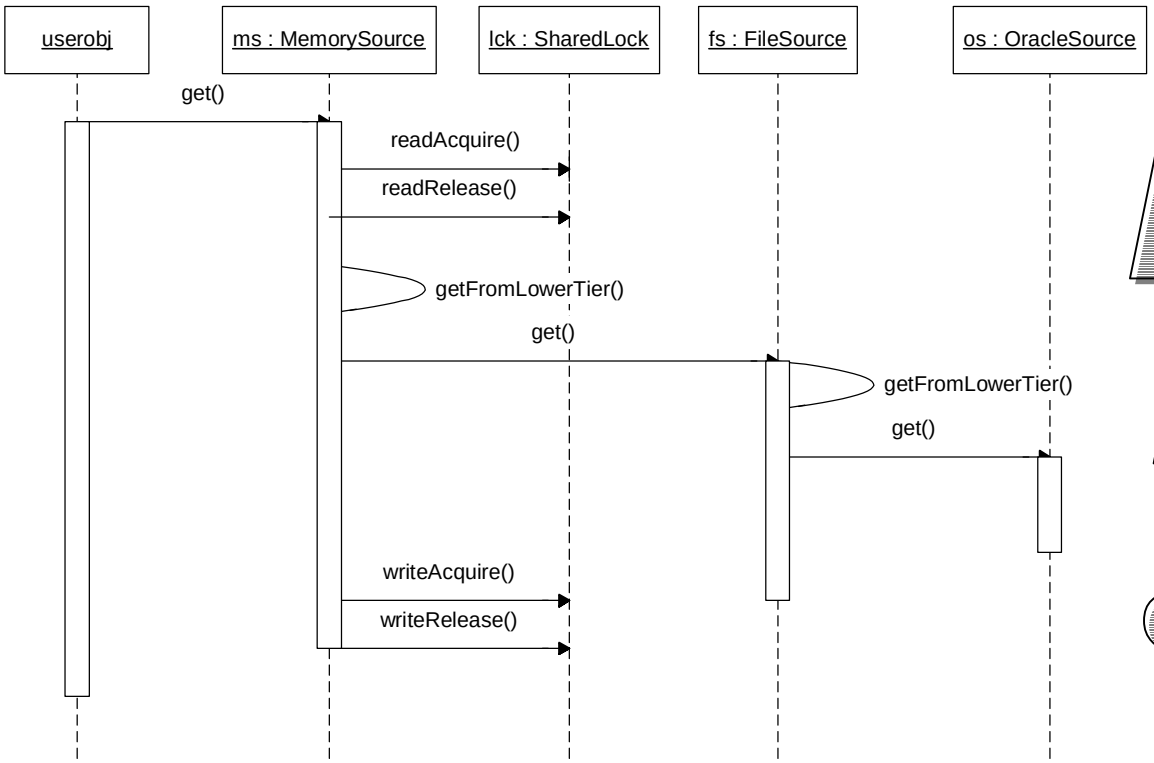
A decision has to be made how the design, errors, and testing procedures will be contained within this document will be maintained. Use of Visio diagrams and reports may be a good strategy.

10 Database Connection Usage



calls

11 Cache Miss Usage



calls